A parallel implementation of tensor multiplication

(CSE260 Project progress report)

Bryan Rasmussen [*]

14 November 2006

1 Overview and Simplifications

The project, as originally conceived, is to develop an efficient parallel code for multiplying tensors up to rank 4. We make several simplifications that tighten the requirements. These should help to make the problem more tractable and will allow for a successful completion of the project within the allotted time.

First, we concentrate exclusively on one type of tensor operation: the product of two rank-4 tensors with two reductions, resulting in another rank-4 tensor. In standard notation, assuming Einstein summation, we write this as

$$r_{abij} = v_{abef} t_{efij}. \tag{1}$$

The motivating application in computational chemistry involves several different types of products with two rank-4 tensors and two contractions, for example,

$$r_{abij} = v_{ijef} t_{abef} \quad \text{and} \quad r_{abij} = v_{afie} t_{bejf}. \tag{2}$$

We could still compute these using the template in Equation (1); we would only have to re-arrange the tensors beforehand. This last point is made easier by the fact that we do not distinguish between covariant and contravariant indices.

The second simplification is that of a 4-*index transformation.* Every rank-4 tensor will, by assumption, have a representation as an element-wise

[*]Los Alamos National Lab; `bryanras@lanl.gov`

sum of products of elements of a matrix, z_{ij}. Specifically, we construct v as

$$v_{abcd} = \sum_{r,s,t,u} z_{ar} z_{bs} z_{ct} z_{du}. \tag{3}$$

This representation can drastically reduce storage requirements at the cost of having to compute individual elements on-the-fly, possibly repeatedly on different processors. Note, too, that the 4-index representation enforces a huge amount of symmetry in the tensor. In general, as long as a, b, c, and d are within the limits on dimension, then $v_{abcd} = v_{cdba} = v_{bdac}$, and in general a specific value is invariant under any permutation of the indices.

2 Current progress

We implement a serial algorithm for tensor multiplication in three ways. The first two examples are `MATLAB` functions, one explicit and one recursive. Both of the functions—which are shown in Sections 6 and 7—are highly inefficient. Their main purpose is illustrative and corrective. That is, they elucidate the basic outline of two different possible algorithms for tensor operations, while also giving a means to check the results of more sophisticated programs.

The explicit routine in Section 6 requires that the arguments be both be rank-4 tensors and that they operate as in Equation (1). It comprises four loops with an internal element-wise multiplication and summation.

The recursive routine in Section 7 is more flexible but even less efficient than the first. The recursive nature of the code requires significantly more dynamic memory allocation and overhead. We include it to demonstrate a different approach to the operation.

A more efficient `MATLAB` routine would employ much more extensive vectorization. It would also be much more difficult to read. Therefore, since the routines are supposed to be pedagogical, we do not seek efficiency here.

The third implementation of the serial algorithm is in C++. This is what we will expand to create the parallel version. It defines a general tensor class called `bryTensor`, which contains several functions, including a multiplication operation with arbitrary reductions and a generalization of the 4-index transformation.

The multiplication operation works with two tensors of arbitrary rank with any number of reductions. The only restriction is that if p is the number of reductions, then the contracted indices be the last p indices of the first tensor and the first p indices of the second tensor. In other words, Equation

(1) is extended to

$$r_{a_1a_2\ldots a_mb_1b_2\ldots b_n} = v_{a_1a_2\ldots a_mc_1c_2\ldots c_p}t_{c_1c_2\ldots c_pb_1b_2\ldots b_n}. \tag{4}$$

We do not use recursive function calls, preferring instead to force a large loop over all possible summations. This costs some additional modulo operations to keep track of current indices, but it is worth it to save memory overhead.

The index transformation code is also more generic than necessary. It assumes that the rank of the tensor is k, with $k > 2$. It computes the tensor from a characteristic matrix, z_{ij}, using the formula

$$v_{i_1i_2\ldots i_k} = \sum_{j_1j_2\cdots j_k} z_{i_1j_1}z_{i_2j_2}\cdots z_{i_kj_k}. \tag{5}$$

All operations under the `bryTensor` class have been checked against the corresponding `MATLAB` code for some small, sample tensors and characteristic matrices. The reader should examine the header file, `bryTensor.h`, and then compile and run the test code, `testen.C`, for details.

3 Strategy

(For the rest of this section, we reference the symbols in Equation (1).)

At first glance, the parallelization of the serial algorithm seems trivial. We know the dimensions of the resulting tensor, r_{abij}, in Equation (1). Moreover, the computation of each component is essentially separate from the computation of the other components.

Assume for concreteness that the dimension of the first index, a, denoted N_a, is the largest dimension in the tensor. We split r into a set of N_a rank-4 tensors, each with dimension one in the first index. (Really, these are rank-three tensors, but it does not cost any extra storage to treat them as rank-4.) We then assign each processor the task of calculating one of these smaller tensors. To do the calculation, a processor needs access to t and $1/N_p$th of v.

Each processor then returns its result to a root "bookkeeper" processor, which probably stores the result on disk. The original processor picks up another task from the root and gets back to work. This strategy assumes that N_a is at least as large as the number of processors - 1, but natural extensions apply if this is not the case.

The difficulty is two-fold. First, it may be beyond the ability of any given processor to store the tensor t, which will necessitate a delicate balance between storage limitations and the additional computational cost of 4-index transformations. (Alternatively, we may store large pieces collectively, but communication overhead makes this unattractive.) Second, there will still be significant communication complexity from having to collect results and distribute pieces of tensors.

The current plan is to write the algorithm using "flat `MPI`". A better method, which we may explore in the future, is to use multi-scale communication—for example `MPI` for large blocks, and then `OpenMP` threads on each processor for sub-components. This architecture will become more salient on machines with multi-core processors or multiple processors per computational node.

One final option is to use a fully-parallel language such as `UPC`. Such a program would of course not be object-oriented, but `UPC`s shared address space model could potentially ease problems with storage and allow for faster execution times. We do not consider this option because of time constraints.

4 Goals and schedule

Because only two weeks remain until the project deadline, it will be necessary to focus efforts on specific cases and live without perfectly optimized code. Three distinct accomplishments are necessary for a successful completion of the project:

1. A working parallel code for the multiplication of two rank-4 tensors with two reductions. Since the serial code already works on arbitrary-rank tensors with an arbitrary number of reductions, it may be just as much work to write general code. We will see.

2. Speedup calculations and scaling information.

3. An understanding of how to improve, expand, and generalize the code.

In addition, several secondary goals would be good to achieve, *only after finishing the first list*. These are less well-defined.

- Take advantage of the gross symmetry from the 4-index transformation to save memory overhead and computation time. This will probably accelerate the code most in the short term.

- Generalize both the serial and parallel versions to handle different types of operations.

- Introduce threads into the parallel version to take advantage of multi-core processors (or just to speed up serial algorithms).
- Investigate the best balance between memory and computation time in the 4-index transformation.
- Clean up the `bryTensor` class, and possibly separate it into multiple classes with an inheritance structure in order to improve extensibility.

To repeat, these are all secondary goals. The bulk of the work will be in obtaining and evaluating a functional and correct parallel code. Hopefully, it will be available within one week (by November 21).

5 Appendix: File list

The following is a list of important files used in the preparation of this report. All files are under in the same directory. The list does not include auxiliary files such as makefiles, etc.

prod2.m Explicit `MATLAB` calculation code listed in Section 6.

prod2rec.m Recursive `MATLAB` calculation code listed in Section 7.

formTen.m `MATLAB` function for creating a tensor from a characteristic matrix.

genTens.m `MATLAB` script for creating some sample tensors.

tensi-j-k-n.dat Text files containing sample tensors. The dimensions are, as the reader probably guessed, i, j, k, and n. The ordering of points in the file is outside-in (row-major for a two-tensor).

charmi-j.dat Text files containing $i \times j$ characteristic matrices.

matlabtens.mat Sample tensors stored in `MATLAB` form.

bryTensor.h Header file for the `bryTensor` class.

bryTensor.C Code for the `bryTensor` class.

testen.C Simple program for testing elements of the `bryTensor` class.

6 Appendix: MATLAB serial algorithm with explicit loops

```
function xx = prod2(vv,ww)

%
% Usage: xx = prod2(vv,ww);
%
% Amazingly inefficient routine for multiplying two
% four-tensors vv and ww with two reductions.
% Assumption of layout is
%
%   xx(ii,jj,kk,ll) = vv(ii,jj,mm,nn)ww(mm,nn,kk,ll);
%
% This routine does no error checking.
%

% Get the dimensions.
Ni = size(vv,1);
Nj = size(vv,2);
Nk = size(ww,3);
Nl = size(ww,4);

xx = zeros(Ni,Nj,Nk,Nl);

% Like I said, this is very inefficient.
for ii=1:Ni
  for jj=1:Nj
    for kk=1:Nk
      for ll=1:Nl

  % Summation.
  temp = squeeze(vv(ii,jj,:,:)).*ww(:,:,kk,ll);
  xx(ii,jj,kk,ll) = sum(temp(:));

      end
    end
  end
end
```

7 Appendix: Recursive MATLAB serial algorithm

```
function xx = prod2rec(vv,ww)

%
% Usage: xx = prod2rec(vv,ww);
%
% Recursive routine for multiplying two tensors
% vv and ww with two reductions. vv and ww can be
% of any rank >= 2.
%
% Assumed format:
%   xx(ii,..,kk,..) = vv(ii,..,mm,nn)ww(mm,nn,kk,..);
%

% If we are already down to the last level, return a scalar.
if (length(size(vv))==2)

  xx = vv.*ww;
  xx=sum(xx(:));

else

  % Strip two dimensions off the tensor with a recursive call.
  Ni = size(vv,1);
  Nk = size(ww,3);
  xdims = length(size(vv))-3;

  % Use Matlab's string evaluations to allow for different
  % ranks in the function call.
  xstring = repmat(',:',1,xdims);

  xeq = ['xx(ii',xstring,',kk',xstring,')'];
  veq = ['vv(ii',xstring,',:,:)'];
  weq = ['ww(:,:,kk',xstring,')'];

  totstring = ...
    [xeq,'=prod2rec(squeeze(',veq,'),squeeze(',weq,'));'];

  % Still very inefficient.
  for ii=1:Ni
    for kk=1:Nk
      eval(totstring);
    end
  end

end
```